# STAA 578 Final Project: Android Malware Classification

Zarah Mattox

## 1 Introduction

Mobile phones are becoming universal, both in the US and globally. In 2022, there were over 3 billion active Android devices worldwide (Samat, 2022). Like all devices, Android devices are vulnerable to malware. Google, the maker of Android, describe malware as, “unsafe or unwanted software that may steal personal info or harm your device,” and suggests keeping devices updated, using higher security settings, and removing untrusted apps, and as a last resort, resetting a device to factory defaults (Google, 2024). Malware can cost users money, expose their personal information, and make devices perform poorly. Users depend on Google’s security features and may consider additional security apps, but malware evolves rapidly and continues to be a risk to Android users.

According to Sharma & Arora, many strategies are used to detect malware, including machine learning and deep learning algorithms. One simple and robust approach is identifying malware based on features used by an app (Sharma & Arora, 2022). Although malware evolves to evade detection, malware’s use of device features is necessary to access device capabilities and personal information. Features analysis is a first line of defense for Android users who want to protect their devices from malware. How well can a deep neural network classify Android malware based on features used? What tradeoffs are there in computational resources and accuracy? Five deep neural networks were used to classify malware based on features used. The models varied widely in computational resources required and resulted in a range of test accuracies between 95% to 96%.

## 2 Method

The Android Malware Detection Dataset, created by Danny Revaldo, contains 4,464 entries with information on 327 features, along with a label of Malware or Benign for each entry. The dataset contains 2,533 entries labeled Malware and 1,931 entries labeled Benign. The 327 features fall into the following categories: Permission, System, Security-related, Communication, Data Access, App Lifecycle, Device Control, and Miscellaneous (Revaldo, 2024). For this dataset, a neural network structured for binary classification was selected. Because this is a binary classification problem a common-sense baseline would be 50% using random guessing.

Jupyter Notebook version 7.0.8 and Python version 3.11.5 with packages keras, tensorflow, numpy, pandas, sklearn, matplotlib were used for analysis. The labels for analysis were scalars where 1 represented Malware and 0 represented Benign. The inputs for analysis were vectors of 0 and 1 indicating which features were used by the app. The dataset was split randomly into a test set that contained 20% of the data and a training set that contained the remaining 80% of the data. Malware proportions for the training and test sets were similar. Five models were compared based on accuracy. A 20% validation split was used for all models.

Models 1, 2, and 3 were optimized using RMSprop. For these models, callbacks were used for early stopping and to create checkpoints. Each model was built based on performance of the previous model. Model 1 was selected as the initial model because it had performed well previously in similar contexts. Because Model 1 performed well again with this dataset, Models 2 and 3 had reduced model complexity to increase speed and reduce computational expense. This provided insight into tradeoffs between accuracy and computational resources when selecting model structure and hyperparameters.

Models 4 and 5 were selected using KerasTuner. KerasTuner selects hyperparameters within a given range. This allows a balance between human expertise and a black box approach to hyperparameter tuning. Again, model accuracy and computational requirements were considered. Model 4 was created using Chapter 13 of *Deep Learning with Python, Second Edition* as a reference (Chollet, 2021). One intermediate layer was selected with range of units from 64 to 512, with a step size of 64. Optimization choices included RMSprop or ADAM. Bayesian optimization was used to tune the model with a maxiumum of 20 trials and 2 executions per trial. Model 5 was created using *Getting started with KerasTuner* (Keras, 2024). The intermediate layers had a range of 1 to 3 and units were tuned separately with a range or 32 to 512 with step size of 32. A 25% Dropout could be added. ADAM was used for optimization with a learning rate in a range of 1e-4 to 1e-2 using log sampling. Random search was used to tune the model with a maximum of 5 trials and 2 executions per trial. Callback was used for both Model 4 and Model 5 for early stopping based on validation loss and a relatively high patience of 5. See code in Appendix for further details, including plots of training and validation accuracy and training and validation loss for each model.

# 3 Outcome

Model 1 contained 5 intermediate layers with units decreasing from 512 by half in each subsequent layer, and 50% dropout between each layer. After 11 epochs, test accuracy was 95.41%. - Among these first three models, Model 1 had highest accuracy.

Model 2 contained 2 intermediate layers with units 32 and 16 and no dropout to check whether Model 1 has more complexity than necessary. After 12 epochs, test accuracy was 94.29%.

Model 3 contained 2 intermediate layers with units 32 and 16, and 50% dropout between each layer. After 11 epochs, test accuracy was 93.17%.

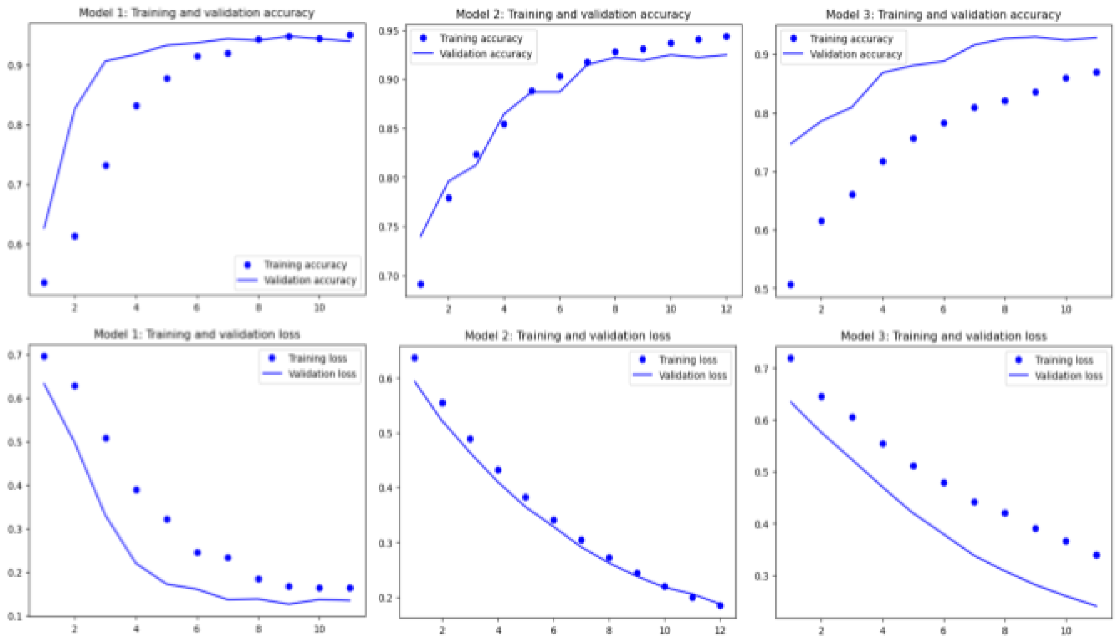



*Figure 1: Plots of training and validation accuracy and training and validation loss for Models 1, 2, and 3. Model 1 performs best among these models.*

For Model 4, KerasTuner selected 64 units for the intermediate layer and RMSprop as the optimizer. KerasTuner took 15 minutes, 44 seconds to run. The tuned model also took slightly longer to run compared to previous models due to more epochs. After 55 epochs, test accuracy was 96.1%. Among models using KerasTuner, Model 4 performed best.

Model 4a was the most computationally intensive process and took over an hour to run due to setting the maximum number of trials to 100. KerasTuner selected 64 units and ADAM as the optimizer. Although this model was due to user error, it is included because it shows that longer run time does not necessarily increase accuracy. Model 4a resulted in test accuracy of 96.0% after 14 epochs.

For Model 5, KerasTuner took 4 minutes 2 seconds to run, which was much faster than Model 4. KerasTuner selected 3 layers with units 192, 32, and 32, respectively. The selected activation was tanh and the selected learning rate for the ADAM optimizer was 0.000826. This model resulted in a test accuracy of 95.6%. After 28 epochs, test accuracy was 96.0%.

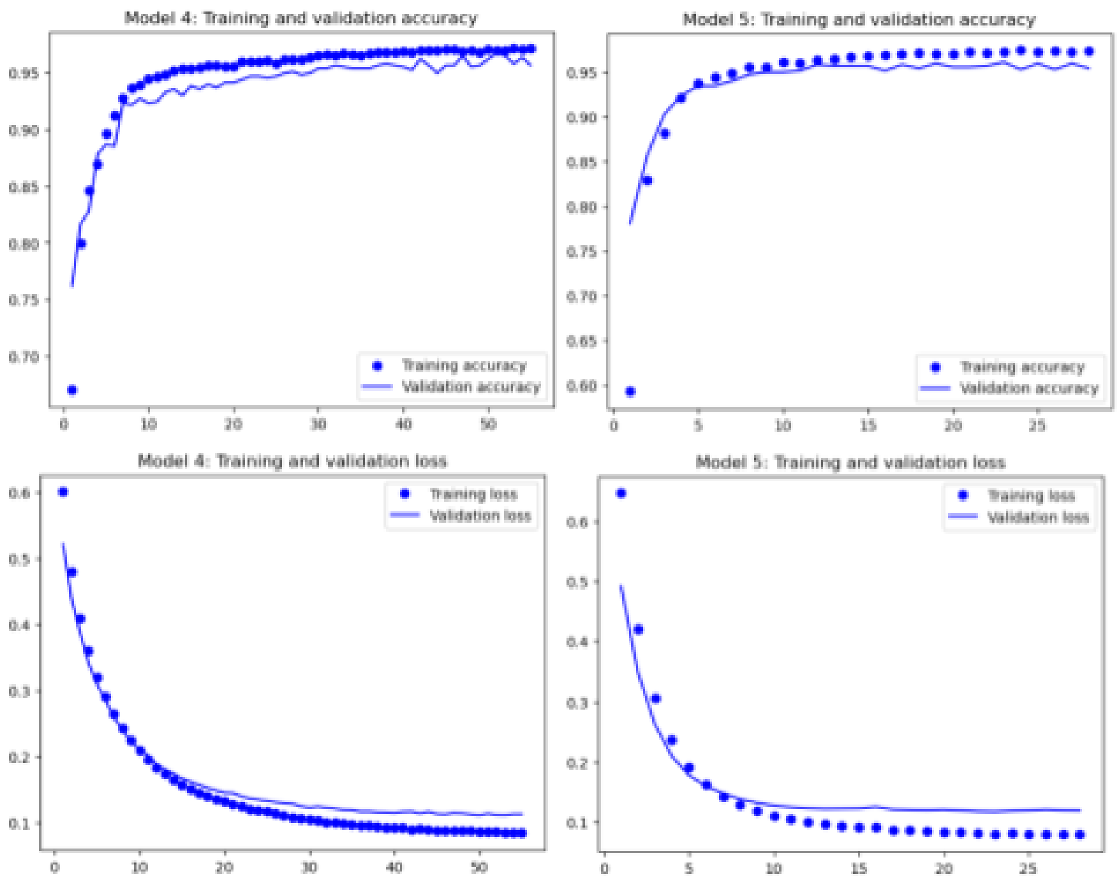



*Figure 2: Plots of training and validation accuracy and training and validation loss for Models 4 and 5. Both models performed comparably well with different optimizers, numbers of layers and units, and numbers of epochs.*

## 4 Discussion

Among all the deep neural network models, there was only a 3% range in test accuracy. Every model performs much better than the common-sense baseline of 50% accuracy. It may be that this particular classification problem is fairly trivial for a deep neural network. Using a "default" model that fits the context and structure of the data (Model 1) is an effective strategy to quickly build an accurate model. Reducing complexity (Models 2 and 3) decreased accuracy and reduced computational expense, but in this case the tradeoff did not support simplifying the model. It is inefficient to spend time trying different models when there is already a model that performs well.

Model 4 and Model 5 are of particular interest in balacing tradeoffs of research time, computational requirements, and accuracy. Using KerasTuner took much guesswork out of hyperparameter tuning and did not require significant computational resources. The tradeoff here is whether the user or KerasTuner spends time and resources selecting hyperparameters. KerasTuner is also useful to compare whether hand-selected hyperparameters perform well. Model 4 performed the best among these models, but Model 1 performed nearly as well. This indicates that a good "default" model is often good enough for many applications. In future deep learning applications, KerasTuner is recommended as a helpful tool to select hyperparameters or provide a baseline of comparison for other models. However, ranges and settings for KerasTuner can quickly become computationally expensive (Model 4a). Judicious option selection based on the dataset and prior experience is necessary for effective use of KerasTuner.

Future research into Android malware classification is recommended. One remaining question is how much feature data is necessary to build a model with a test accuracy of 95%. Can such high accuracy be achieved with a lower number of entries? Another question is whether specific feature categories better predict whether an Android application is malware. Perhaps a few permissions or a particular category of features accounts for most of the predictive power of these deep neural network models. Next, Android malware classification is often composed of a variety of techniques, so research into ensembles of models may result in higher classification accuracy. Android malware changes rapidly, so techniques that are effective today need to be constantly reassessed and improved upon.

# 5 References


Chollet, F. (2021). *Deep Learning with Python, Second Edition.* Manning Publications

Google. (2024, May 8). Remove unwanted ads, pop-ups & malware. *Google Chrome Help*. https://support.google.com/chrome/answer/2765944

Keras. (2024, May 8). Keras Documentation: Getting started with Kerastuner. *Keras Developer Guides*. https://keras.io/guides/keras_tuner/getting_started/

Revaldo, D. (2024, February). Android Malware Detection Dataset, Version 1. Retrieved May 2, 2024 from https://www.kaggle.com/datasets/dannyrevaldo/android-malware-detection-dataset

Samat, S. (2022, May 11). Living in a multi-device world with Android. *Google*. https://blog.google/products/android/io22-multideviceworld/

Sharma, Y., & Arora, A. (2024). A comprehensive review on permissions-based Android malware detection. *International Journal of Information Security*. https://doi.org/10.1007/s10207-024-00822-2


# 6 Appendix

## 6.1 Prepare Data

[1]:
```
from tensorflow import keras
keras.utils.set_random_seed(578)
import numpy as np
import pandas as pd
```

```
2024-05-09 08:43:56.402540: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 AVX_VNNI FMA, in
other operations, rebuild TensorFlow with the appropriate compiler flags.
```

[2]:
```python
# import dataset
df = pd.read_csv("Android_Malware_Benign.csv", header=0)
```

[3]:
```python
# 4464 entries, 328 variables
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4464 entries, 0 to 4463
Columns: 328 entries, ACCESS_ALL_DOWNLOADS to Label
dtypes: int64(327), object(1)
memory usage: 11.2+ MB
```

[4]:
```python
# Look at variables
variables = list(df)
# variables # save list of variables, but it is long and not displayed here
```

[5]:
```python
## Look at first five rows to get an understanding of the structure
## This output is not included because there are 328 columns.
## df[0:5]
```

[6]:
```python
# Label information
df['Label'].info()
```

```
<class 'pandas.core.series.Series'>
RangeIndex: 4464 entries, 0 to 4463
Series name: Label
Non-Null Count  Dtype
--------------  -----
4464 non-null   object
dtypes: object(1)
memory usage: 35.0+ KB
```

[7]:
```python
# What are the labels?
df['Label'].unique()
```

[7]:
```
array(['Malware', 'Benign'], dtype=object)
```

[8]:
```python
# How many are Malware?
sum(df['Label']=='Malware')
```

[8]:
```
2533
```

[9]:
```python
# How many are Benign?
sum(df['Label']!='Malware')
```

[9]:
```
1931
```

[10]:
```
# Create y where 1 is Malware and 0 is Benign
y = pd.get_dummies(df['Label'], drop_first=True)
```

[11]:
```
y.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4464 entries, 0 to 4463
Data columns (total 1 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Malware  4464 non-null   bool
dtypes: bool(1)
memory usage: 4.5 KB
```

[12]:
```
y[0:5]
```

[12]:

| | Malware |
|---|---|
| 0 | True |
| 1 | True |
| 2 | True |
| 3 | True |
| 4 | True |

[13]:
```
# Value counts match 'Labels'
y.value_counts()
```

[13]:
```
Malware
True     2533
False    1931
Name: count, dtype: int64
```

[14]:
```
x=df.drop(columns=['Label'])
x = x.astype("bool")
```

[15]:
```
## Verify that x contains appropriate data
## x[0:5]
x.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4464 entries, 0 to 4463
Columns: 327 entries, ACCESS_ALL_DOWNLOADS to
android.permission.PROCESS_INCOMING_CALLS
dtypes: bool(327)
memory usage: 1.4 MB
```

## 6.2 Split the data into training and test sets

[16]:
```
from sklearn.model_selection import train_test_split
# for reproducibility
# x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,␣
↪random_state=578)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
```

[17]:
```
print(f"Proportion malware in full dataset: {round(np.mean(y_train.values),4)}")
print(f"Proportion malware in test set: {round(np.mean(y_test.values),4)}")
```

```
Proportion malware in full dataset: 0.5707
Proportion malware in test set: 0.5543
```

## 6.3 Build Models

[18]:
```
from tensorflow import keras
from tensorflow.keras import layers
keras.mixed_precision.set_global_policy("mixed_float16")
keras.utils.set_random_seed(578)
```

```
WARNING:tensorflow:Mixed precision compatibility check (mixed_float16): WARNING
The dtype policy mixed_float16 may run slowly because this machine does not have
a GPU. Only Nvidia GPUs with compute capability of at least 7.0 run quickly with
mixed_float16.
If you will use compatible GPU(s) not attached to this host, e.g. by running a
multi-worker model, you can ignore this warning. This message will only be
logged once
```

### 6.3.1 Model 1

[19]:
```
# Model 1
callbacks_list1 = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=2,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath="model1.keras",
        monitor="val_loss",
        save_best_only=True,
    )
]

model = keras.Sequential([
layers.Dense(512, activation='relu', input_shape=(x_train.shape[1],)),
layers.Dropout(0.5),
layers.Dense(256, activation='relu'),
```

```
layers.Dropout(0.5),
layers.Dense(128, activation='relu'),
layers.Dropout(0.5),
layers.Dense(64, activation='relu'),
layers.Dropout(0.5),
layers.Dense(32, activation='relu'),
layers.Dropout(0.5),
layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer="rmsprop", loss='binary_crossentropy',␣
↪metrics=['accuracy'])
```

```
2024-05-09 08:43:57.909119: I
tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool
with default inter op setting: 2. Tune using inter_op_parallelism_threads for
best performance.
```

[20]:

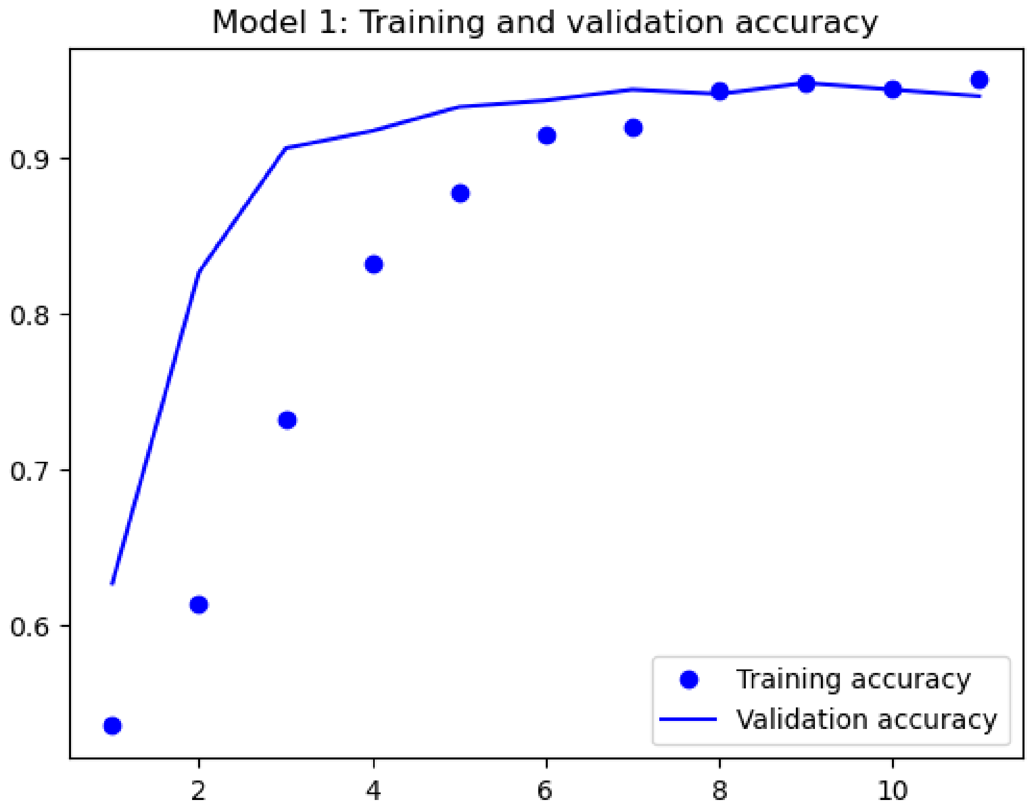
```
history = model.fit(
    x_train, y_train,
    epochs=100,
    batch_size=512,
    validation_split=0.2,
    verbose=2,
    callbacks=callbacks_list1)
```

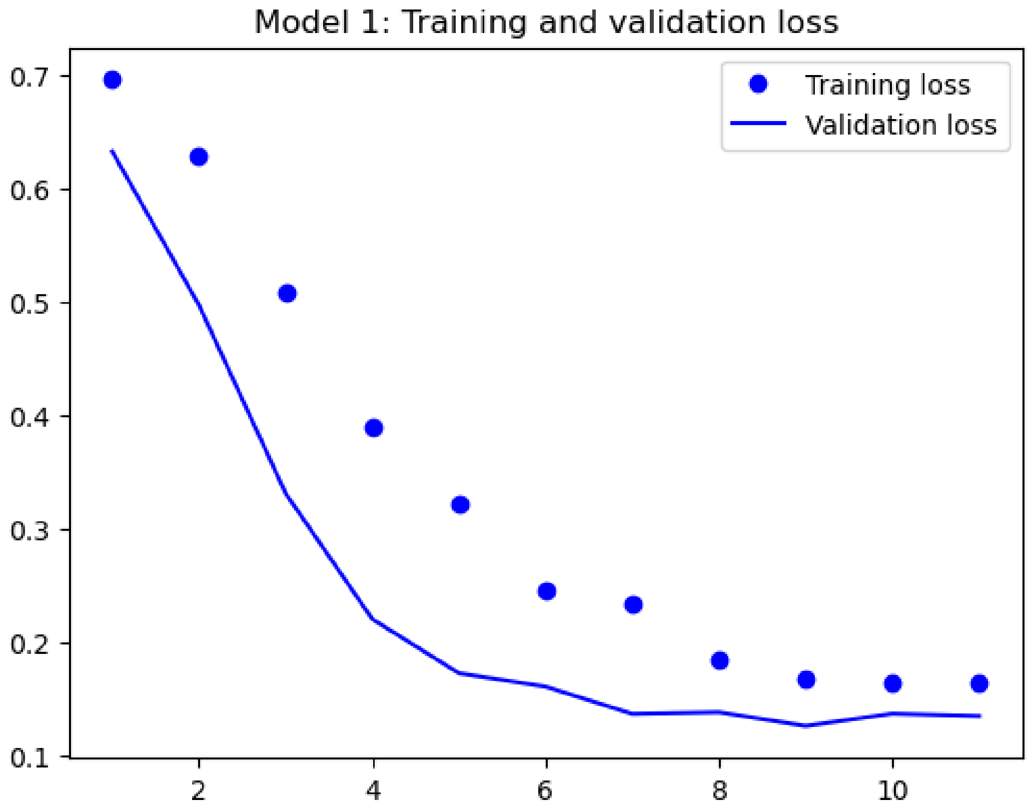
```
Epoch 1/100
6/6 - 4s - loss: 0.6956 - accuracy: 0.5350 - val_loss: 0.6325 - val_accuracy:
0.6266 - 4s/epoch - 587ms/step
Epoch 2/100
6/6 - 3s - loss: 0.6280 - accuracy: 0.6134 - val_loss: 0.4972 - val_accuracy:
0.8266 - 3s/epoch - 474ms/step
Epoch 3/100
6/6 - 3s - loss: 0.5091 - accuracy: 0.7318 - val_loss: 0.3312 - val_accuracy:
0.9063 - 3s/epoch - 484ms/step
Epoch 4/100
6/6 - 3s - loss: 0.3892 - accuracy: 0.8326 - val_loss: 0.2204 - val_accuracy:
0.9175 - 3s/epoch - 493ms/step
Epoch 5/100
6/6 - 3s - loss: 0.3215 - accuracy: 0.8778 - val_loss: 0.1726 - val_accuracy:
0.9329 - 3s/epoch - 492ms/step
Epoch 6/100
6/6 - 3s - loss: 0.2451 - accuracy: 0.9149 - val_loss: 0.1608 - val_accuracy:
0.9371 - 3s/epoch - 504ms/step
Epoch 7/100
6/6 - 3s - loss: 0.2342 - accuracy: 0.9202 - val_loss: 0.1368 - val_accuracy:
0.9441 - 3s/epoch - 514ms/step
Epoch 8/100
6/6 - 3s - loss: 0.1855 - accuracy: 0.9429 - val_loss: 0.1382 - val_accuracy:
```

```
0.9413 - 3s/epoch - 516ms/step
Epoch 9/100
6/6 - 4s - loss: 0.1679 - accuracy: 0.9478 - val_loss: 0.1262 - val_accuracy:
0.9483 - 4s/epoch - 731ms/step
Epoch 10/100
6/6 - 4s - loss: 0.1638 - accuracy: 0.9447 - val_loss: 0.1370 - val_accuracy:
0.9441 - 4s/epoch - 743ms/step
Epoch 11/100
6/6 - 5s - loss: 0.1650 - accuracy: 0.9503 - val_loss: 0.1349 - val_accuracy:
0.9399 - 5s/epoch - 764ms/step
```

[21]:

```
import matplotlib.pyplot as plt
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)
plt.plot(epochs, accuracy, "bo", label="Training accuracy")
plt.plot(epochs, val_accuracy, "b", label="Validation accuracy")
plt.title("Model 1: Training and validation accuracy")
plt.legend()
plt.figure()
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Model 1: Training and validation loss")
plt.legend()
plt.show()
```


Model 1: Training and validation accuracy
0.9
0.8
0.7
0.6
2
4
6
8
10
Training accuracy
Validation accuracy




[22]:
```
## Evaluating the model
test_model = keras.models.load_model("model1.keras")
test_loss, test_acc = test_model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.3f}")
```

```
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your
model is starting with a freshly initialized optimizer.
28/28 [==============================] - 1s 20ms/step - loss: 0.1298 - accuracy:
0.9541
Test accuracy: 0.954
```

### 6.3.2 Model 2

[23]:
```
# Model 2
callbacks_list2 = [
keras.callbacks.EarlyStopping(
monitor="val_accuracy",
patience=2,
),
keras.callbacks.ModelCheckpoint(
filepath="model2.keras",
```

```
monitor="val_loss",
save_best_only=True,
)
]

model = keras.Sequential([
layers.Dense(32, activation='relu'),
layers.Dense(16, activation='relu'),
layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer="rmsprop", loss='binary_crossentropy',␣
↪metrics=['accuracy'])
```

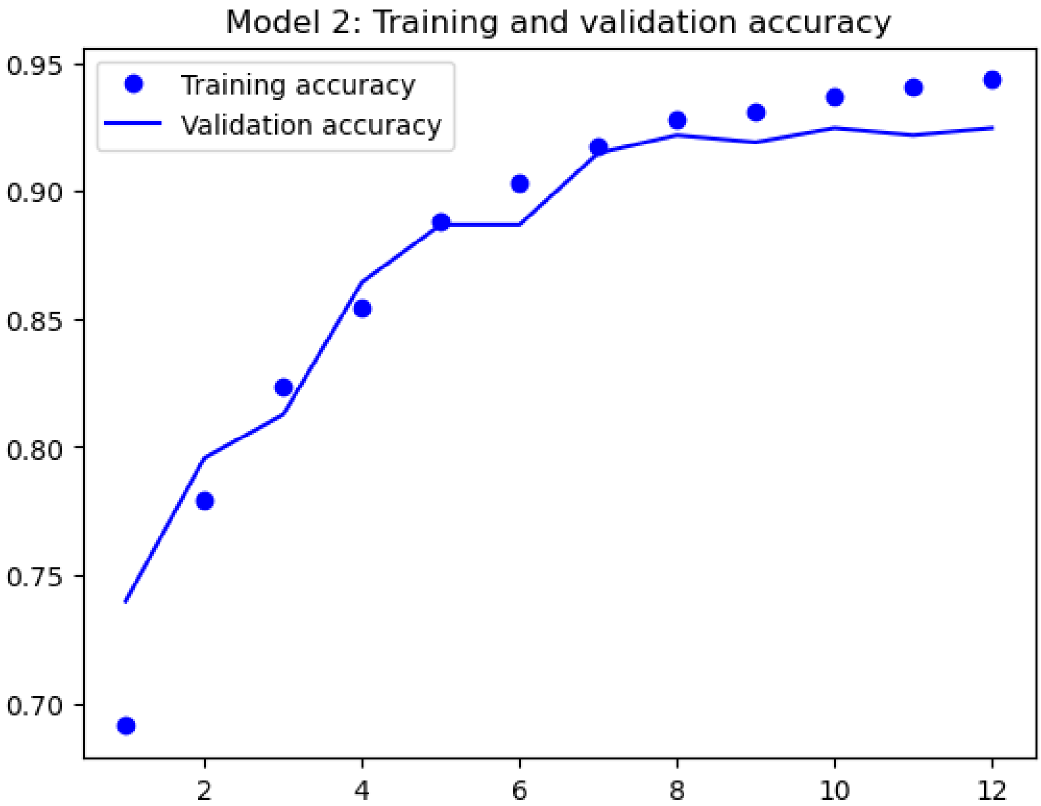
[24]:
```
history = model.fit(
    x_train, y_train,
    epochs=100,
    batch_size=512,
    validation_split=0.2,
    verbose=2,
    callbacks=callbacks_list2)
```

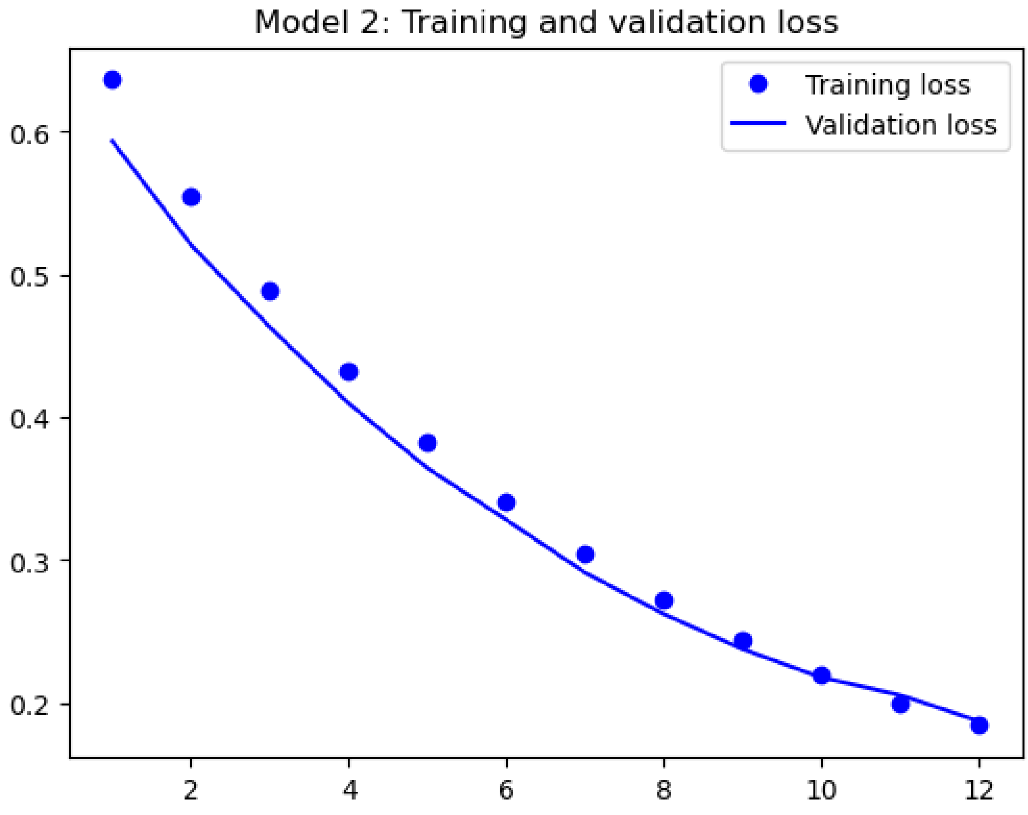
```
Epoch 1/100
6/6 - 1s - loss: 0.6363 - accuracy: 0.6912 - val_loss: 0.5934 - val_accuracy:
0.7399 - 628ms/epoch - 105ms/step
Epoch 2/100
6/6 - 0s - loss: 0.5546 - accuracy: 0.7794 - val_loss: 0.5207 - val_accuracy:
0.7958 - 162ms/epoch - 27ms/step
Epoch 3/100
6/6 - 0s - loss: 0.4886 - accuracy: 0.8239 - val_loss: 0.4631 - val_accuracy:
0.8126 - 171ms/epoch - 29ms/step
Epoch 4/100
6/6 - 0s - loss: 0.4330 - accuracy: 0.8547 - val_loss: 0.4097 - val_accuracy:
0.8643 - 171ms/epoch - 29ms/step
Epoch 5/100
6/6 - 0s - loss: 0.3831 - accuracy: 0.8883 - val_loss: 0.3643 - val_accuracy:
0.8867 - 170ms/epoch - 28ms/step
Epoch 6/100
6/6 - 0s - loss: 0.3413 - accuracy: 0.9034 - val_loss: 0.3281 - val_accuracy:
0.8867 - 185ms/epoch - 31ms/step
Epoch 7/100
6/6 - 0s - loss: 0.3041 - accuracy: 0.9174 - val_loss: 0.2912 - val_accuracy:
0.9147 - 194ms/epoch - 32ms/step
Epoch 8/100
6/6 - 0s - loss: 0.2717 - accuracy: 0.9282 - val_loss: 0.2621 - val_accuracy:
0.9217 - 159ms/epoch - 27ms/step
Epoch 9/100
6/6 - 0s - loss: 0.2439 - accuracy: 0.9310 - val_loss: 0.2376 - val_accuracy:
```

```
0.9189 - 177ms/epoch - 29ms/step
Epoch 10/100
6/6 - 0s - loss: 0.2202 - accuracy: 0.9366 - val_loss: 0.2177 - val_accuracy:
0.9245 - 135ms/epoch - 22ms/step
Epoch 11/100
6/6 - 0s - loss: 0.1997 - accuracy: 0.9405 - val_loss: 0.2055 - val_accuracy:
0.9217 - 163ms/epoch - 27ms/step
Epoch 12/100
6/6 - 0s - loss: 0.1841 - accuracy: 0.9433 - val_loss: 0.1871 - val_accuracy:
0.9245 - 144ms/epoch - 24ms/step
```

[25]:

```
import matplotlib.pyplot as plt
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)
plt.plot(epochs, accuracy, "bo", label="Training accuracy")
plt.plot(epochs, val_accuracy, "b", label="Validation accuracy")
plt.title("Model 2: Training and validation accuracy")
plt.legend()
plt.figure()
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Model 2: Training and validation loss")
plt.legend()
plt.show()
```


Model 2: Training and validation accuracy
Training accuracy
Validation accuracy
0.95
0.90
0.85
0.80
0.75
0.70
2
4
6
8
10
12




[26]:
```
## Evaluating the model
test_model = keras.models.load_model("model2.keras")
test_loss, test_acc = test_model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.3f}")
```

```
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your
model is starting with a freshly initialized optimizer.
28/28 [==============================] - 0s 2ms/step - loss: 0.1771 - accuracy:
0.9429
Test accuracy: 0.943
```

### 6.3.3 Model 3

[27]:
```
# Model 3
callbacks_list3 = [
keras.callbacks.EarlyStopping(
monitor="val_accuracy",
patience=2,
),
keras.callbacks.ModelCheckpoint(
filepath="model3.keras",
```

```
monitor="val_loss",
save_best_only=True,
)
]

model = keras.Sequential([
layers.Dense(32, activation='relu'),
layers.Dropout(0.5),
layers.Dense(16, activation='relu'),
layers.Dropout(0.5),
layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer="rmsprop", loss='binary_crossentropy',␣
↪metrics=['accuracy'])
```

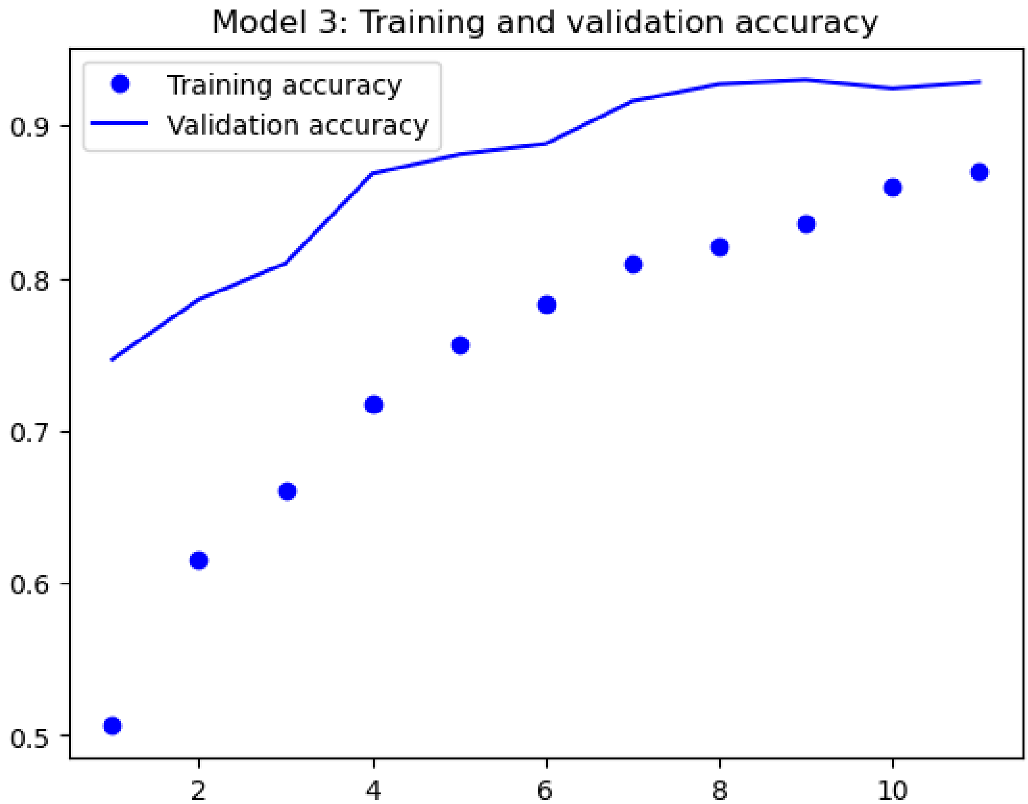
[28]:
```
history = model.fit(
    x_train, y_train,
    epochs=100,
    batch_size=512,
    validation_split=0.2,
    verbose=2,
    callbacks=callbacks_list3)
```

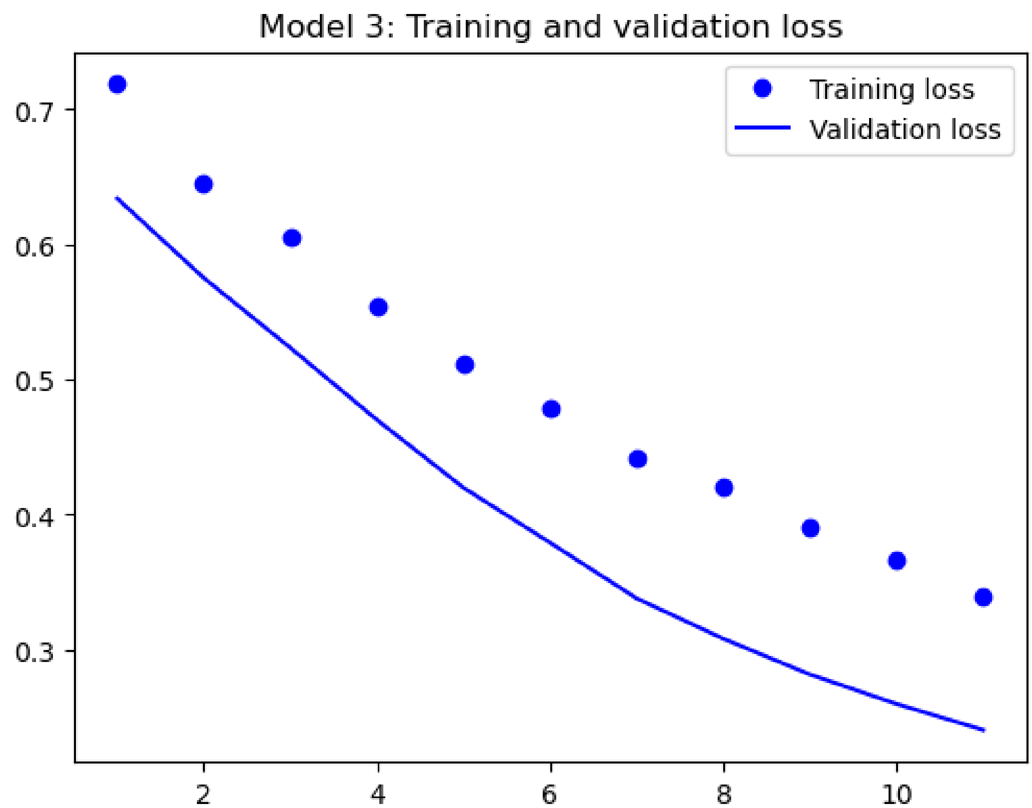
```
Epoch 1/100
6/6 - 1s - loss: 0.7185 - accuracy: 0.5063 - val_loss: 0.6341 - val_accuracy:
0.7469 - 633ms/epoch - 106ms/step
Epoch 2/100
6/6 - 0s - loss: 0.6448 - accuracy: 0.6159 - val_loss: 0.5752 - val_accuracy:
0.7860 - 155ms/epoch - 26ms/step
Epoch 3/100
6/6 - 0s - loss: 0.6060 - accuracy: 0.6607 - val_loss: 0.5234 - val_accuracy:
0.8098 - 165ms/epoch - 27ms/step
Epoch 4/100
6/6 - 0s - loss: 0.5542 - accuracy: 0.7171 - val_loss: 0.4701 - val_accuracy:
0.8685 - 174ms/epoch - 29ms/step
Epoch 5/100
6/6 - 0s - loss: 0.5120 - accuracy: 0.7563 - val_loss: 0.4198 - val_accuracy:
0.8811 - 194ms/epoch - 32ms/step
Epoch 6/100
6/6 - 0s - loss: 0.4783 - accuracy: 0.7833 - val_loss: 0.3791 - val_accuracy:
0.8881 - 197ms/epoch - 33ms/step
Epoch 7/100
6/6 - 0s - loss: 0.4418 - accuracy: 0.8102 - val_loss: 0.3379 - val_accuracy:
0.9161 - 163ms/epoch - 27ms/step
Epoch 8/100
6/6 - 0s - loss: 0.4204 - accuracy: 0.8204 - val_loss: 0.3081 - val_accuracy:
0.9273 - 185ms/epoch - 31ms/step
```

```
Epoch 9/100
6/6 - 0s - loss: 0.3899 - accuracy: 0.8365 - val_loss: 0.2816 - val_accuracy:
0.9301 - 190ms/epoch - 32ms/step
Epoch 10/100
6/6 - 0s - loss: 0.3664 - accuracy: 0.8599 - val_loss: 0.2598 - val_accuracy:
0.9245 - 168ms/epoch - 28ms/step
Epoch 11/100
6/6 - 0s - loss: 0.3391 - accuracy: 0.8701 - val_loss: 0.2405 - val_accuracy:
0.9287 - 164ms/epoch - 27ms/step
```

[29]:

```
import matplotlib.pyplot as plt
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)
plt.plot(epochs, accuracy, "bo", label="Training accuracy")
plt.plot(epochs, val_accuracy, "b", label="Validation accuracy")
plt.title("Model 3: Training and validation accuracy")
plt.legend()
plt.figure()
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Model 3: Training and validation loss")
plt.legend()
plt.show()
```


Model 3: Training and validation accuracy
Training accuracy
Validation accuracy
0.9
0.8
0.7
0.6
0.5
2
4
6
8
10




[30]:
```
## Evaluating the model
test_model = keras.models.load_model("model3.keras")
test_loss, test_acc = test_model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.3f}")
```

```
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your
model is starting with a freshly initialized optimizer.
28/28 [==============================] - 0s 3ms/step - loss: 0.2337 - accuracy:
0.9317
Test accuracy: 0.932
```

### 6.3.4 Model 4: Exploring KerasTuner

[31]:
```
# p. 414 in textbook
import keras_tuner as kt
```

[32]:
```
def build_model(hp):
    units = hp.Int(name="units", min_value=64, max_value=512, step=64)
    model = keras.Sequential([
        layers.Dense(units, activation="relu"),
        layers.Dense(1, activation="sigmoid")
```

```
    ])
    optimizer = hp.Choice(name="optimizer", values=["rmsprop", "adam"])
    model.compile(
        optimizer=optimizer,
        loss="binary_crossentropy",
        metrics=["accuracy"])
    return model
```

[33]:
```
tuner = kt.BayesianOptimization(
    build_model,
    objective="val_accuracy",
    max_trials=20,
    executions_per_trial=2,
    directory="kt_test",
    overwrite=True,
)
```

[34]:
```
tuner.search_space_summary()
```

```
Search space summary
Default search space size: 2
units (Int)
{'default': None, 'conditions': [], 'min_value': 64, 'max_value': 512, 'step':
64, 'sampling': 'linear'}
optimizer (Choice)
{'default': 'rmsprop', 'conditions': [], 'values': ['rmsprop', 'adam'],
'ordered': False}
```

[35]:
```
callbacks=[keras.callbacks.EarlyStopping(monitor="val_loss", patience=5)]
```

[36]:
```
tuner.search(
    x_train, y_train,
    batch_size=128,
    epochs=100,
    validation_split=0.2,
    callbacks=callbacks,
    verbose=2,
)
```

```
Trial 20 Complete [00h 00m 35s]
val_accuracy: 0.9622377753257751

Best val_accuracy So Far: 0.9678321778774261
Total elapsed time: 00h 16m 11s
```

Trial 100 Complete [00h 00m 22s] val_accuracy: 0.9594405591487885

Best val_accuracy So Far: 0.9650349617004395 Total elapsed time: 01h 16m 45s

[37]:
```
tuner.results_summary(4)
```

```
Results summary
Results in kt_test/untitled_project
Showing 4 best trials
Objective(name="val_accuracy", direction="max")

Trial 14 summary
Hyperparameters:
units: 64
optimizer: rmsprop
Score: 0.9678321778774261

Trial 12 summary
Hyperparameters:
units: 64
optimizer: rmsprop
Score: 0.966433584690094

Trial 13 summary
Hyperparameters:
units: 128
optimizer: rmsprop
Score: 0.9636363387107849

Trial 15 summary
Hyperparameters:
units: 384
optimizer: adam
Score: 0.9636363387107849
```

[38]:
```
# Get the top 4 hyperparameters.
top_n = 4
best_hps = tuner.get_best_hyperparameters(top_n)
# Build the model with the best hp.
model = build_model(best_hps[0])

history = model.fit(
    x_train, y_train,
    epochs=100,
    batch_size=512,
    validation_split=0.2,
    verbose=2,
    callbacks=callbacks)
```

```
Epoch 1/100
6/6 - 1s - loss: 0.6013 - accuracy: 0.6698 - val_loss: 0.5215 - val_accuracy:
0.7622 - 803ms/epoch - 134ms/step
```

```
Epoch 2/100
6/6 - 0s - loss: 0.4800 - accuracy: 0.7994 - val_loss: 0.4399 - val_accuracy:
0.8168 - 278ms/epoch - 46ms/step
Epoch 3/100
6/6 - 0s - loss: 0.4092 - accuracy: 0.8463 - val_loss: 0.3882 - val_accuracy:
0.8280 - 275ms/epoch - 46ms/step
Epoch 4/100
6/6 - 0s - loss: 0.3606 - accuracy: 0.8697 - val_loss: 0.3423 - val_accuracy:
0.8783 - 280ms/epoch - 47ms/step
Epoch 5/100
6/6 - 0s - loss: 0.3209 - accuracy: 0.8964 - val_loss: 0.3094 - val_accuracy:
0.8867 - 294ms/epoch - 49ms/step
Epoch 6/100
6/6 - 0s - loss: 0.2913 - accuracy: 0.9128 - val_loss: 0.2856 - val_accuracy:
0.8853 - 289ms/epoch - 48ms/step
Epoch 7/100
6/6 - 0s - loss: 0.2655 - accuracy: 0.9275 - val_loss: 0.2599 - val_accuracy:
0.9231 - 286ms/epoch - 48ms/step
Epoch 8/100
6/6 - 0s - loss: 0.2436 - accuracy: 0.9366 - val_loss: 0.2404 - val_accuracy:
0.9217 - 318ms/epoch - 53ms/step
Epoch 9/100
6/6 - 0s - loss: 0.2255 - accuracy: 0.9398 - val_loss: 0.2232 - val_accuracy:
0.9273 - 326ms/epoch - 54ms/step
Epoch 10/100
6/6 - 0s - loss: 0.2096 - accuracy: 0.9450 - val_loss: 0.2104 - val_accuracy:
0.9231 - 348ms/epoch - 58ms/step
Epoch 11/100
6/6 - 0s - loss: 0.1958 - accuracy: 0.9464 - val_loss: 0.2020 - val_accuracy:
0.9245 - 350ms/epoch - 58ms/step
Epoch 12/100
6/6 - 0s - loss: 0.1843 - accuracy: 0.9482 - val_loss: 0.1884 - val_accuracy:
0.9329 - 294ms/epoch - 49ms/step
Epoch 13/100
6/6 - 0s - loss: 0.1740 - accuracy: 0.9524 - val_loss: 0.1810 - val_accuracy:
0.9357 - 258ms/epoch - 43ms/step
Epoch 14/100
6/6 - 0s - loss: 0.1655 - accuracy: 0.9534 - val_loss: 0.1746 - val_accuracy:
0.9301 - 256ms/epoch - 43ms/step
Epoch 15/100
6/6 - 0s - loss: 0.1580 - accuracy: 0.9541 - val_loss: 0.1673 - val_accuracy:
0.9385 - 251ms/epoch - 42ms/step
Epoch 16/100
6/6 - 0s - loss: 0.1512 - accuracy: 0.9548 - val_loss: 0.1622 - val_accuracy:
0.9357 - 249ms/epoch - 42ms/step
Epoch 17/100
6/6 - 0s - loss: 0.1451 - accuracy: 0.9566 - val_loss: 0.1577 - val_accuracy:
0.9399 - 249ms/epoch - 41ms/step
```

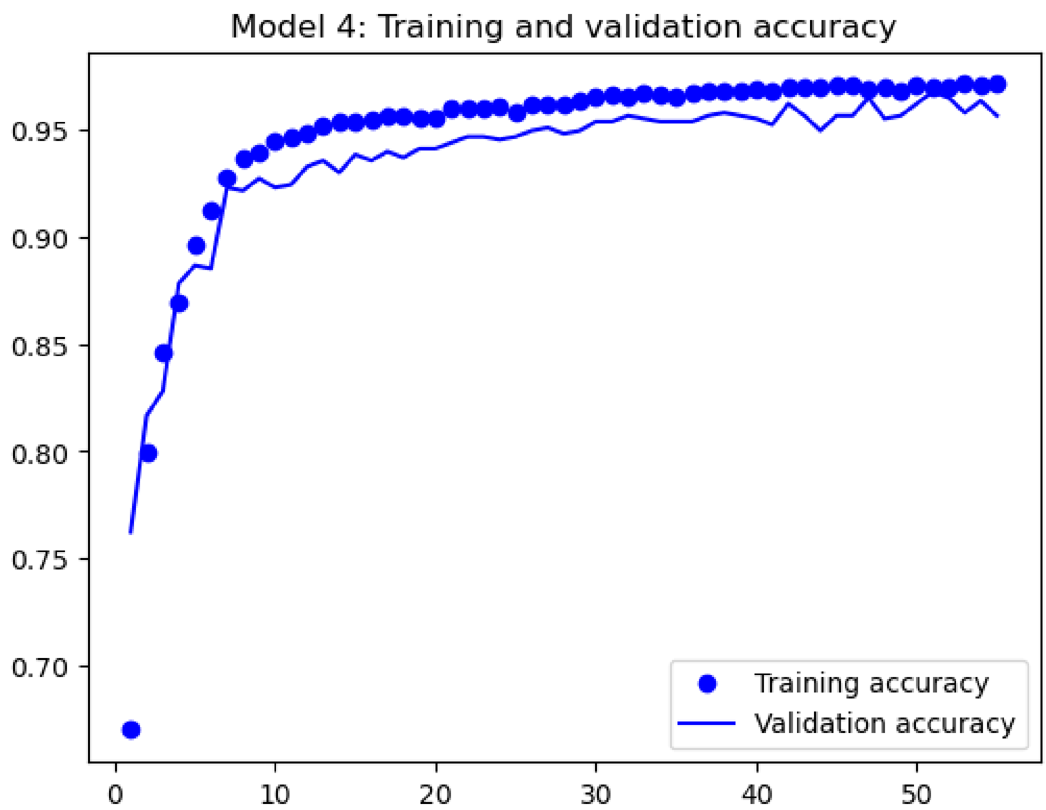
```
Epoch 18/100
6/6 - 0s - loss: 0.1411 - accuracy: 0.9562 - val_loss: 0.1528 - val_accuracy:
0.9371 - 257ms/epoch - 43ms/step
Epoch 19/100
6/6 - 0s - loss: 0.1358 - accuracy: 0.9552 - val_loss: 0.1494 - val_accuracy:
0.9413 - 265ms/epoch - 44ms/step
Epoch 20/100
6/6 - 0s - loss: 0.1327 - accuracy: 0.9555 - val_loss: 0.1457 - val_accuracy:
0.9413 - 273ms/epoch - 45ms/step
Epoch 21/100
6/6 - 0s - loss: 0.1284 - accuracy: 0.9597 - val_loss: 0.1452 - val_accuracy:
0.9441 - 308ms/epoch - 51ms/step
Epoch 22/100
6/6 - 0s - loss: 0.1246 - accuracy: 0.9601 - val_loss: 0.1402 - val_accuracy:
0.9469 - 301ms/epoch - 50ms/step
Epoch 23/100
6/6 - 0s - loss: 0.1209 - accuracy: 0.9604 - val_loss: 0.1370 - val_accuracy:
0.9469 - 296ms/epoch - 49ms/step
Epoch 24/100
6/6 - 0s - loss: 0.1184 - accuracy: 0.9611 - val_loss: 0.1352 - val_accuracy:
0.9455 - 277ms/epoch - 46ms/step
Epoch 25/100
6/6 - 0s - loss: 0.1168 - accuracy: 0.9583 - val_loss: 0.1334 - val_accuracy:
0.9469 - 287ms/epoch - 48ms/step
Epoch 26/100
6/6 - 0s - loss: 0.1136 - accuracy: 0.9622 - val_loss: 0.1314 - val_accuracy:
0.9497 - 296ms/epoch - 49ms/step
Epoch 27/100
6/6 - 0s - loss: 0.1107 - accuracy: 0.9622 - val_loss: 0.1298 - val_accuracy:
0.9510 - 289ms/epoch - 48ms/step
Epoch 28/100
6/6 - 0s - loss: 0.1086 - accuracy: 0.9615 - val_loss: 0.1290 - val_accuracy:
0.9483 - 308ms/epoch - 51ms/step
Epoch 29/100
6/6 - 0s - loss: 0.1070 - accuracy: 0.9636 - val_loss: 0.1250 - val_accuracy:
0.9497 - 304ms/epoch - 51ms/step
Epoch 30/100
6/6 - 0s - loss: 0.1044 - accuracy: 0.9657 - val_loss: 0.1233 - val_accuracy:
0.9538 - 291ms/epoch - 49ms/step
Epoch 31/100
6/6 - 0s - loss: 0.1029 - accuracy: 0.9664 - val_loss: 0.1246 - val_accuracy:
0.9538 - 285ms/epoch - 48ms/step
Epoch 32/100
6/6 - 0s - loss: 0.1012 - accuracy: 0.9657 - val_loss: 0.1231 - val_accuracy:
0.9566 - 283ms/epoch - 47ms/step
Epoch 33/100
6/6 - 0s - loss: 0.1002 - accuracy: 0.9671 - val_loss: 0.1216 - val_accuracy:
0.9552 - 294ms/epoch - 49ms/step
```

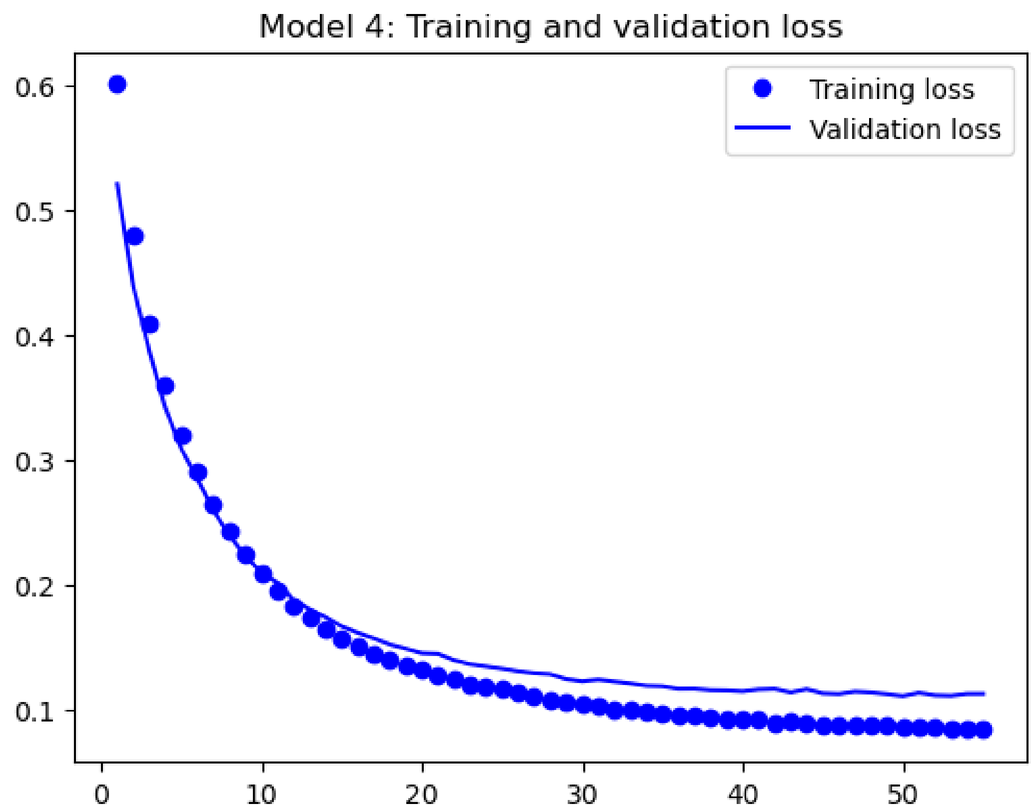
```
Epoch 34/100
6/6 - 0s - loss: 0.0983 - accuracy: 0.9660 - val_loss: 0.1197 - val_accuracy:
0.9538 - 281ms/epoch - 47ms/step
Epoch 35/100
6/6 - 0s - loss: 0.0971 - accuracy: 0.9657 - val_loss: 0.1193 - val_accuracy:
0.9538 - 281ms/epoch - 47ms/step
Epoch 36/100
6/6 - 0s - loss: 0.0959 - accuracy: 0.9674 - val_loss: 0.1172 - val_accuracy:
0.9538 - 277ms/epoch - 46ms/step
Epoch 37/100
6/6 - 0s - loss: 0.0952 - accuracy: 0.9681 - val_loss: 0.1173 - val_accuracy:
0.9566 - 273ms/epoch - 45ms/step
Epoch 38/100
6/6 - 0s - loss: 0.0941 - accuracy: 0.9678 - val_loss: 0.1162 - val_accuracy:
0.9580 - 268ms/epoch - 45ms/step
Epoch 39/100
6/6 - 0s - loss: 0.0935 - accuracy: 0.9678 - val_loss: 0.1160 - val_accuracy:
0.9566 - 281ms/epoch - 47ms/step
Epoch 40/100
6/6 - 0s - loss: 0.0922 - accuracy: 0.9688 - val_loss: 0.1153 - val_accuracy:
0.9552 - 280ms/epoch - 47ms/step
Epoch 41/100
6/6 - 0s - loss: 0.0923 - accuracy: 0.9681 - val_loss: 0.1167 - val_accuracy:
0.9524 - 274ms/epoch - 46ms/step
Epoch 42/100
6/6 - 0s - loss: 0.0902 - accuracy: 0.9699 - val_loss: 0.1172 - val_accuracy:
0.9622 - 288ms/epoch - 48ms/step
Epoch 43/100
6/6 - 0s - loss: 0.0918 - accuracy: 0.9699 - val_loss: 0.1144 - val_accuracy:
0.9566 - 286ms/epoch - 48ms/step
Epoch 44/100
6/6 - 0s - loss: 0.0897 - accuracy: 0.9695 - val_loss: 0.1169 - val_accuracy:
0.9497 - 280ms/epoch - 47ms/step
Epoch 45/100
6/6 - 0s - loss: 0.0887 - accuracy: 0.9706 - val_loss: 0.1136 - val_accuracy:
0.9566 - 268ms/epoch - 45ms/step
Epoch 46/100
6/6 - 0s - loss: 0.0887 - accuracy: 0.9706 - val_loss: 0.1131 - val_accuracy:
0.9566 - 268ms/epoch - 45ms/step
Epoch 47/100
6/6 - 0s - loss: 0.0876 - accuracy: 0.9688 - val_loss: 0.1150 - val_accuracy:
0.9650 - 321ms/epoch - 53ms/step
Epoch 48/100
6/6 - 0s - loss: 0.0875 - accuracy: 0.9702 - val_loss: 0.1144 - val_accuracy:
0.9552 - 322ms/epoch - 54ms/step
Epoch 49/100
6/6 - 0s - loss: 0.0875 - accuracy: 0.9685 - val_loss: 0.1131 - val_accuracy:
0.9566 - 309ms/epoch - 52ms/step
```

```
Epoch 50/100
6/6 - 0s - loss: 0.0866 - accuracy: 0.9709 - val_loss: 0.1115 - val_accuracy:
0.9622 - 288ms/epoch - 48ms/step
Epoch 51/100
6/6 - 0s - loss: 0.0861 - accuracy: 0.9695 - val_loss: 0.1141 - val_accuracy:
0.9678 - 280ms/epoch - 47ms/step
Epoch 52/100
6/6 - 0s - loss: 0.0864 - accuracy: 0.9702 - val_loss: 0.1121 - val_accuracy:
0.9650 - 273ms/epoch - 46ms/step
Epoch 53/100
6/6 - 0s - loss: 0.0849 - accuracy: 0.9713 - val_loss: 0.1118 - val_accuracy:
0.9580 - 269ms/epoch - 45ms/step
Epoch 54/100
6/6 - 0s - loss: 0.0843 - accuracy: 0.9709 - val_loss: 0.1133 - val_accuracy:
0.9636 - 279ms/epoch - 46ms/step
Epoch 55/100
6/6 - 0s - loss: 0.0849 - accuracy: 0.9713 - val_loss: 0.1133 - val_accuracy:
0.9566 - 245ms/epoch - 41ms/step
```

[39]:

```python
import matplotlib.pyplot as plt
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)
plt.plot(epochs, accuracy, "bo", label="Training accuracy")
plt.plot(epochs, val_accuracy, "b", label="Validation accuracy")
plt.title("Model 4: Training and validation accuracy")
plt.legend()
plt.figure()
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Model 4: Training and validation loss")
plt.legend()
plt.show()
```


Model 4: Training and validation accuracy
0.95
0.90
0.85
0.80
0.75
0.70
0
10
20
30
40
50
Training accuracy
Validation accuracy




[40]:
```
## Evaluating the model
test_model = model
test_loss, test_acc = test_model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.3f}")
```

```
28/28 [==============================] - 0s 3ms/step - loss: 0.1236 - accuracy:
0.9608
Test accuracy: 0.961
```

#### 6.3.5 Model 4a: max_trials=100

**Model 4a test accuracy results from previous setting max_trials=100** Epoch 14/100 6/6 - 0s - loss: 0.0880 - accuracy: 0.9674 - val_loss: 0.1227 - val_accuracy: 0.9580 - 296ms/epoch - 49ms/step WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your model is starting with a freshly initialized optimizer. 28/28 [==============================] - 0s 7ms/step - loss: 0.1239 - accuracy: 0.9597 Test accuracy: 0.960

### 6.3.6 Model 5

[41]:
```
## Model 5
# https://keras.io/guides/keras_tuner/getting_started/
import keras_tuner
def build_model(hp):
    model = keras.Sequential()
    # Tune the number of layers.
    for i in range(hp.Int("num_layers", 1, 3)):
        model.add(
            layers.Dense(
                # Tune number of units separately.
                units=hp.Int(f"units_{i}", min_value=32, max_value=512,␣
 ↪step=32),
                activation=hp.Choice("activation", ["relu", "tanh"]),
            )
        )
    if hp.Boolean("dropout"):
        model.add(layers.Dropout(rate=0.25))
    model.add(layers.Dense(1, activation="sigmoid"))
    learning_rate = hp.Float("lr", min_value=1e-4, max_value=1e-2,␣
 ↪sampling="log")
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
        loss="binary_crossentropy",
        metrics=["accuracy"],
    )
    return model


build_model(keras_tuner.HyperParameters())
```

[41]:
```
<keras.engine.sequential.Sequential at 0x784ea1fda740>
```

[42]:
```
tuner = keras_tuner.RandomSearch(
    hypermodel=build_model,
    objective="val_accuracy",
    max_trials=5,
    executions_per_trial=2,
    overwrite=True,
    directory="kt2_test",
)
```

[43]:
```
tuner.search(
    x_train, y_train,
    epochs=100,
    validation_split=0.2,
```

```
    callbacks=callbacks,
    verbose=2,
)
```

```
Trial 5 Complete [00h 00m 36s]
val_accuracy: 0.9636363387107849

Best val_accuracy So Far: 0.9636363387107849
Total elapsed time: 00h 08m 56s
```

[44]:
```
tuner.results_summary()
```

```
Results summary
Results in kt2_test/untitled_project
Showing 10 best trials
Objective(name="val_accuracy", direction="max")

Trial 1 summary
Hyperparameters:
num_layers: 3
units_0: 192
activation: tanh
dropout: False
lr: 0.0008259900546842376
units_1: 32
units_2: 32
Score: 0.9636363387107849

Trial 4 summary
Hyperparameters:
num_layers: 1
units_0: 224
activation: tanh
dropout: False
lr: 0.001958975633214816
units_1: 192
units_2: 480
Score: 0.9636363387107849

Trial 0 summary
Hyperparameters:
num_layers: 1
units_0: 448
activation: tanh
dropout: False
lr: 0.00012840079280248366
Score: 0.96293705701828
```

```
Trial 2 summary
Hyperparameters:
num_layers: 2
units_0: 32
activation: relu
dropout: False
lr: 0.00032679946602554217
units_1: 384
units_2: 160
Score: 0.9622377753257751

Trial 3 summary
Hyperparameters:
num_layers: 3
units_0: 480
activation: tanh
dropout: False
lr: 0.0011331682914451785
units_1: 512
units_2: 512
Score: 0.9615384638309479
```

[45]:

```
# Get the top 2 hyperparameters.
top_n = 2
best_hps = tuner.get_best_hyperparameters(top_n)
# Build the model with the best hp.
model = build_model(best_hps[0])

history = model.fit(
    x_train, y_train,
    epochs=100,
    batch_size=512,
    validation_split=0.2,
    verbose=2,
    callbacks=callbacks)
```

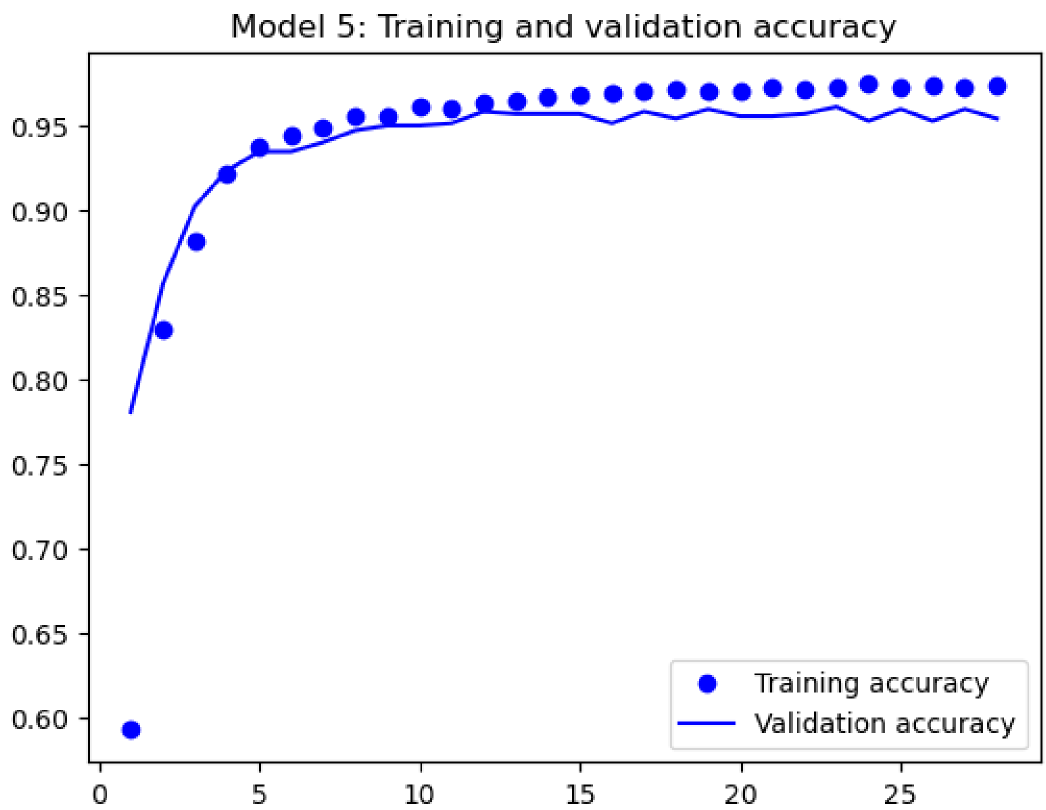
```
Epoch 1/100
6/6 - 1s - loss: 0.6463 - accuracy: 0.5924 - val_loss: 0.4918 - val_accuracy:
0.7804 - 1s/epoch - 217ms/step
Epoch 2/100
6/6 - 1s - loss: 0.4201 - accuracy: 0.8295 - val_loss: 0.3483 - val_accuracy:
0.8559 - 624ms/epoch - 104ms/step
Epoch 3/100
6/6 - 1s - loss: 0.3068 - accuracy: 0.8810 - val_loss: 0.2618 - val_accuracy:
0.9021 - 614ms/epoch - 102ms/step
Epoch 4/100
6/6 - 1s - loss: 0.2360 - accuracy: 0.9209 - val_loss: 0.2091 - val_accuracy:
0.9231 - 621ms/epoch - 104ms/step
```

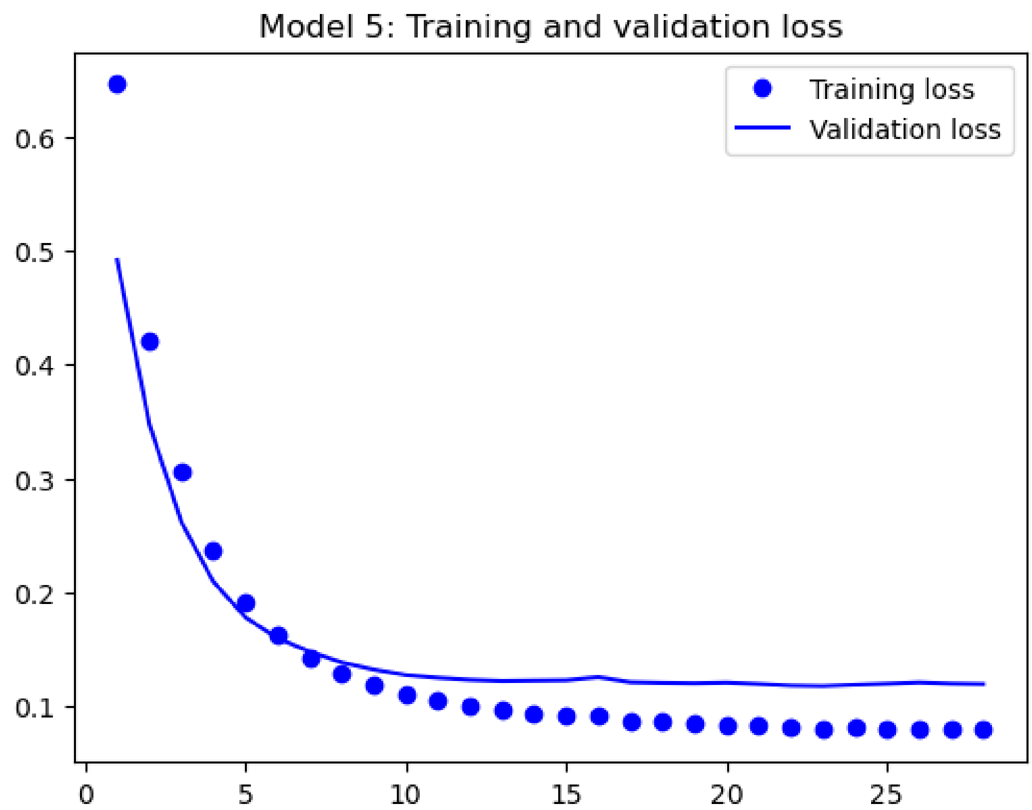
```
Epoch 5/100
6/6 - 1s - loss: 0.1905 - accuracy: 0.9366 - val_loss: 0.1778 - val_accuracy:
0.9343 - 637ms/epoch - 106ms/step
Epoch 6/100
6/6 - 1s - loss: 0.1624 - accuracy: 0.9436 - val_loss: 0.1593 - val_accuracy:
0.9343 - 637ms/epoch - 106ms/step
Epoch 7/100
6/6 - 1s - loss: 0.1419 - accuracy: 0.9482 - val_loss: 0.1477 - val_accuracy:
0.9399 - 651ms/epoch - 108ms/step
Epoch 8/100
6/6 - 1s - loss: 0.1283 - accuracy: 0.9559 - val_loss: 0.1382 - val_accuracy:
0.9469 - 1s/epoch - 176ms/step
Epoch 9/100
6/6 - 1s - loss: 0.1188 - accuracy: 0.9552 - val_loss: 0.1319 - val_accuracy:
0.9497 - 1s/epoch - 243ms/step
Epoch 10/100
6/6 - 2s - loss: 0.1104 - accuracy: 0.9611 - val_loss: 0.1271 - val_accuracy:
0.9497 - 2s/epoch - 253ms/step
Epoch 11/100
6/6 - 1s - loss: 0.1046 - accuracy: 0.9601 - val_loss: 0.1247 - val_accuracy:
0.9510 - 1s/epoch - 242ms/step
Epoch 12/100
6/6 - 1s - loss: 0.0993 - accuracy: 0.9632 - val_loss: 0.1229 - val_accuracy:
0.9580 - 1s/epoch - 243ms/step
Epoch 13/100
6/6 - 1s - loss: 0.0962 - accuracy: 0.9646 - val_loss: 0.1219 - val_accuracy:
0.9566 - 1s/epoch - 244ms/step
Epoch 14/100
6/6 - 1s - loss: 0.0930 - accuracy: 0.9664 - val_loss: 0.1222 - val_accuracy:
0.9566 - 1s/epoch - 246ms/step
Epoch 15/100
6/6 - 2s - loss: 0.0923 - accuracy: 0.9681 - val_loss: 0.1225 - val_accuracy:
0.9566 - 2s/epoch - 250ms/step
Epoch 16/100
6/6 - 2s - loss: 0.0914 - accuracy: 0.9692 - val_loss: 0.1252 - val_accuracy:
0.9510 - 2s/epoch - 257ms/step
Epoch 17/100
6/6 - 1s - loss: 0.0872 - accuracy: 0.9699 - val_loss: 0.1208 - val_accuracy:
0.9580 - 1s/epoch - 244ms/step
Epoch 18/100
6/6 - 1s - loss: 0.0856 - accuracy: 0.9709 - val_loss: 0.1203 - val_accuracy:
0.9538 - 1s/epoch - 183ms/step
Epoch 19/100
6/6 - 1s - loss: 0.0846 - accuracy: 0.9702 - val_loss: 0.1199 - val_accuracy:
0.9594 - 857ms/epoch - 143ms/step
Epoch 20/100
6/6 - 1s - loss: 0.0833 - accuracy: 0.9706 - val_loss: 0.1205 - val_accuracy:
0.9552 - 818ms/epoch - 136ms/step
```

```
Epoch 21/100
6/6 - 1s - loss: 0.0825 - accuracy: 0.9720 - val_loss: 0.1192 - val_accuracy:
0.9552 - 851ms/epoch - 142ms/step
Epoch 22/100
6/6 - 1s - loss: 0.0809 - accuracy: 0.9716 - val_loss: 0.1178 - val_accuracy:
0.9566 - 771ms/epoch - 128ms/step
Epoch 23/100
6/6 - 1s - loss: 0.0803 - accuracy: 0.9727 - val_loss: 0.1174 - val_accuracy:
0.9608 - 771ms/epoch - 128ms/step
Epoch 24/100
6/6 - 1s - loss: 0.0806 - accuracy: 0.9741 - val_loss: 0.1185 - val_accuracy:
0.9524 - 813ms/epoch - 135ms/step
Epoch 25/100
6/6 - 1s - loss: 0.0795 - accuracy: 0.9720 - val_loss: 0.1195 - val_accuracy:
0.9594 - 724ms/epoch - 121ms/step
Epoch 26/100
6/6 - 1s - loss: 0.0803 - accuracy: 0.9737 - val_loss: 0.1206 - val_accuracy:
0.9524 - 759ms/epoch - 127ms/step
Epoch 27/100
6/6 - 1s - loss: 0.0794 - accuracy: 0.9720 - val_loss: 0.1196 - val_accuracy:
0.9594 - 732ms/epoch - 122ms/step
Epoch 28/100
6/6 - 1s - loss: 0.0788 - accuracy: 0.9730 - val_loss: 0.1192 - val_accuracy:
0.9538 - 767ms/epoch - 128ms/step
```

[46]:

```
import matplotlib.pyplot as plt
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)
plt.plot(epochs, accuracy, "bo", label="Training accuracy")
plt.plot(epochs, val_accuracy, "b", label="Validation accuracy")
plt.title("Model 5: Training and validation accuracy")
plt.legend()
plt.figure()
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Model 5: Training and validation loss")
plt.legend()
plt.show()
```


Model 5: Training and validation accuracy
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0
5
10
15
20
25
Training accuracy
Validation accuracy




[47]:
```
## Evaluating the model
test_model = model
test_loss, test_acc = test_model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.3f}")
```

```
28/28 [==============================] - 0s 9ms/step - loss: 0.1259 - accuracy:
0.9597
Test accuracy: 0.960
```